Part 9: Stochastic Gradient Descent

February 1, 2021

We are going to go over a more light hearted topic, that of stochastic gradient descent (SGD). The basic idea is as follows, if we are solving an optimization problem where solving for the gradient is difficult, usually in the case of doing large matrix inverses, we should use SGD. This often happens because (i) the gradient is composed of that large matrix inverse or is otherwise computationally difficult (ii) the gradient is composed of randomly distributed parameters.

1 Basic SGD

We need something to compare SGD to, so we will solve the ordinary least squares method (OLS) problem.

$$\beta_{OLS} = (X^T X)^{-1} X^T Y$$

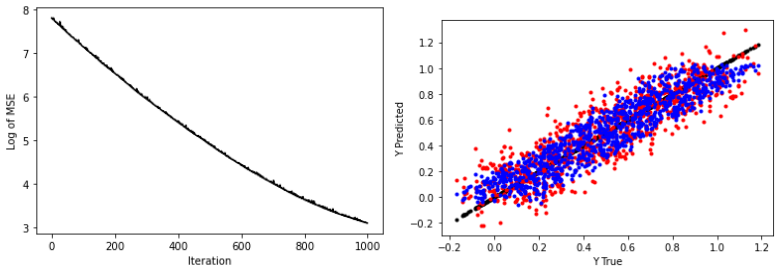
Where X will be a $N \times p$ matrix with $N = 10^3$ and $p = 10^2$ and $Y = X_1 + U(-0.2, 0.2)$. Of course this is not a hard problem for python to solve on most computers. However, we could imagine harder problems such as medical databases having many more data points N. We could also imagine p being much larger for images (remember that a typical image might have $p = 360 \times 360$ pixels).

For SGD we do normal gradient descent, but when we calculate the gradient we only use a random selection of points $\{X, Y\}$. In the full gradient descent of the least squares problem the gradient is as follows:

$$f(\beta) = \Sigma_i^N (Y_i - \beta X_i^T)^2$$

$$\nabla f(\beta) = \Sigma_i^N 2(Y_i - \beta X_i^T) X_i$$

Which is the likelihood of the least squares problem, but for SGD we will only use B random selections of $\{X, Y\}$. Therefore, for this application of SGD, we are using point (i), where the matrix X is large. Note also that $\nabla f(\beta)$ is the gradient of the loss function, which can be used in a standard gradient descent update rule.



(a) SGD Multiple Iterations (b) Error for OLS and SGD

Figure 1: SGD for Solving Large Matrix Computation | blue = OLS, red = SGD

Here we will extend SGD to point (ii), where the problem is not large but itself stochastic. Let us have the same problem above except that $N = 10^3$ and $p = 10$ (a smaller problem) but let's assume the coefficients of the linear model are stochastically described by independent normal distributions.

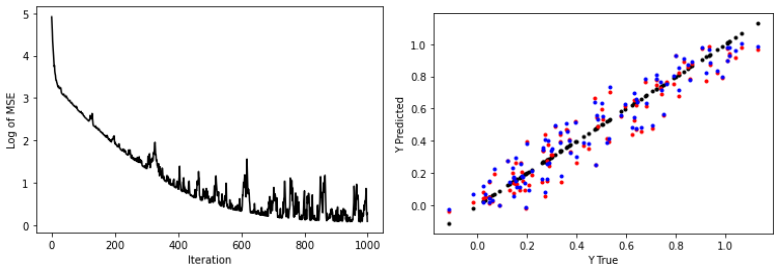
$$\beta \sim N(\mu, I \times \sigma)$$

Here we will use **all** data $\{X, Y\}$ when computing the likelihood $f(\beta)$, but we will do that process B times using the above generating distribution. The trick to the update rule will be to have $\mu \approx \beta$ in the SGD loop. If you are confused look into the python code provided.

This second problem, to me, seems much truer to form when it comes to SGD. We are literally calculating $\nabla f(\beta)$ B times then taking an average, whereas in the first problem we aren't technically calculating the correct gradient to the deterministic problem. Either way we solve the problem efficiently.

2 Advanced SGD

Here we will discuss some more advanced versions of SGD, namely (i) ADAGRAD, (ii) RMSPROP, and (iii) ADAM. We will test all of them on the least squares problem with $N = 100$ and $p = 10$ for computational simplicity, but obviously such small problems could be solved with OLS.



(a) SGD Multiple Iterations (b) Error for OLS and SGD

Figure 2: SGD for Solving Stochastic Parameter Problems | blue = OLS, red = SGD

2.1 ADAGRAD

The first advanced SGD is ADAGRAD, where we literally add the L2 norm of all previous gradient calculations for each parameter and divide the current gradient $\nabla f(\beta)$ by this weighting. This way larger parameters get dampened and those with smaller learning rates get artificially increased.

$$G_{jj} = \Sigma_t^T g_{t,j}^2$$

$$\beta_j = \beta_j - \eta \frac{\nabla_j f(\beta)}{\sqrt{G_{jj}}}$$

The result, as well as all results for advanced methods, is shown in Figure 2

2.2 RMSPROP

Next we have RMSPROP, or root-mean-squared propagation where we normalize the learning rate by a dampened L2 norm of the gradient. That dampened gradient is as follows:

$$v_{t+1} = \gamma v_t + (1 - \gamma)(\nabla f(\beta))^2$$

Now we take that dampened gradient and use that as a normalization factor for η.

$$\beta = \beta - \eta \frac{\nabla f(\beta)}{\sqrt{v}}$$

2.3 ADAM

This last one is ADAM, or adaptive moment estimation. Here we take our gradient of the loss function $\nabla f(\beta)$ and modify it as such:

$$m = \alpha_1 m + (1 - \alpha_1)\nabla f(\beta)$$

Where m acts as our new gradient modulated by a "forgetting factor" α_1. We do the same with the square of the gradient:

$$v = \alpha_2 v + (1 - \alpha_2)(\nabla f(\beta)^2)$$

Which is the adjustment v for the square of the loss gradient. We now calculate an effective gradient and squared loss gradient:

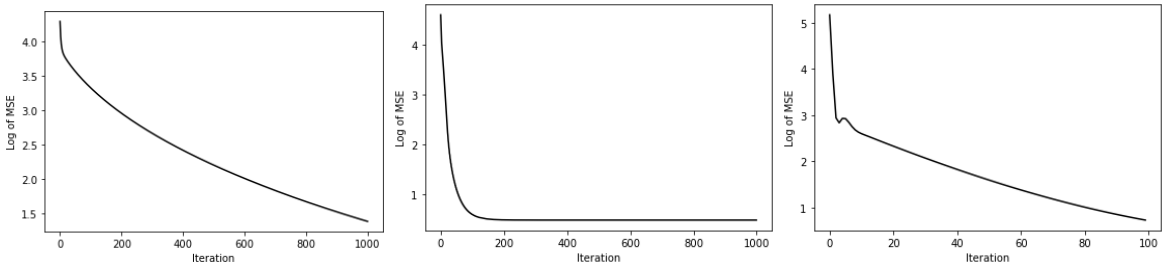
$$\hat{m} = \frac{m}{1 - \alpha_1^{t+1}}$$

$$\hat{v} = \frac{v}{1 - \alpha_2^{t+1}}$$

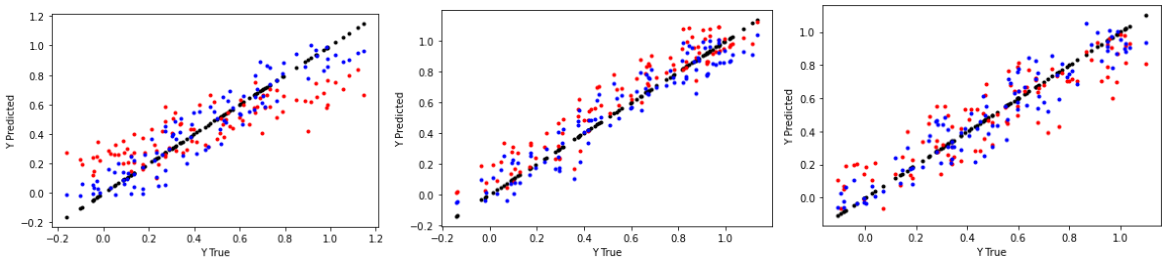
We get the update rule as such:

$$\beta = \beta + \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

Where $\epsilon = 10^{-8}$, $\alpha_1 = 0.9$, and $\alpha_2 = 0.999$.



(a) ADAGRAD (b) RMSPROP (c) ADAM



(d) ADAGRAD (e) RMSPROP (f) ADAM

Figure 3: Advanced SGD Algorithms | blue = OLS, red = SGD